Corrigé TP2

1. Toute formule du calcul propositionnel est équivalente à une formule sous forme normale conjonctive.
2. C'est immédiat d'après les lois de De Morgan :

$$\begin{aligned}(b_1 \wedge ... \wedge b_m) \Rightarrow (a_1 \vee ... \vee a_n) &\equiv \neg(b_1 \wedge ... \wedge b_m) \vee (a_1 \vee ... \vee a_n) \\ &\equiv \neg b_1 \vee ... \vee \neg b_m \vee a_1 ... \vee a_n\end{aligned}$$

   et cette formule est bien la clause dont les a_i sont les variables positives et les b_i les négatives.

3. Les clauses C_1 et C_2 sont supposées simplifiées. Pour obtenir la clause en résultant par coupure sur v, on commence par supprimer (l'unique occurrence de) v de C_1 et $-v$ de C_2 avec supprime_variable. Puis, on fusionne les deux listes obtenues - par hypothèse triées et sans doublons - en une liste triée et sans doublons (pour tenir compte de l'opération de simplification) avec fusion.

```
let coupure (c1:clause) (c2:clause) (v:int) :clause =
  let rec supprime_variable (c:clause) (v:int) :clause =
       match c with
       |[] -> []
       |t::q when (t = v) -> supprime_variable q v
       |t::q -> t::(supprime_variable q v)
  in
  let rec fusion (c1:clause) (c2:clause) :clause =
    match c1, c2 with
      |[], _ -> c2
      |c1, [] -> c1
      |t1::q1, t2::q2 when t1 < t2 -> t1::(fusion q1 (t2::q2))
      |t1::q1, t2::q2 when t1 = t2 -> t1::(fusion q1 q2)
      |t1::q1, t2::q2 -> t2::(fusion (t1::q1) q2)
  in fusion (supprime_variable c1 v) (supprime_variable c2 (-v))
```

   Remarquez qu'il est en fait inutile d'appeler supprime_variable récursivement dans le deuxième cas du filtrage puisque l'énoncé précise que les clauses sont simplifiées par défaut donc une variable y apparaît au plus une fois.

4. Les variables recherchées sont les éléments de C_1 dont l'opposé est dans C_2. La liste obtenue sera sans doublon puique C_1 est supposée simplifiée par défaut.

```
let rec variables_a_couper (c1:clause) (c2:clause) :int list =
  match c1  with
    |[] -> []
    |t::q -> if List.mem (-t) c2 then t::(variables_a_couper q c2)
             else variables_a_couper q c2
```

5. On construit l'ensemble des variables par coupure sur lesquelles on peut déduire une clause de C_1 et de C_2 avec variables_a_couper. Pour chacune d'elles, on effectue la coupure incriminée avec coupure :

```
let nouvelles_clauses (c1:clause) (c2:clause) :clause list =
  List.map (fun v -> (coupure c1 c2 v)) (variables_a_couper c1 c2)
```

6. Il s'agit juste de faire un test d'appartenance adapté :

```
let exists_clause_vide (lc:clause list) :bool = List.mem [] lc
```

7. On introduit une fonction traitement prenant en entrée deux listes de clauses C_t et C_p et appliquant récursivement les opérations décrites par l'énoncé.

   Si il n'y a plus de clause à traiter ($C_t = \emptyset$), on renvoie la liste C_p des clauses prouvables. Sinon, on récupère le premier élément t de C_t. On construit toutes les clauses qu'il est possible de déduire à partir de t et d'une des clauses de C_p à l'aide de nouvelles_clauses. On purge l'ensemble de clauses obtenues de celles qui sont dans C_t ou C_p avec List.filter pour obtenir un ensemble E. Il ne reste plus qu'à appliquer récursivement ce traitement en ajoutant aux clauses à traiter toutes celles de E et en faisant basculer t dans les clauses prouvables.

   Une fois les clauses prouvables à partir de l'ensemble de clauses initial obtenu à l'aide de traitement, on vérifie s'il contient la clause vide avec exists_clause_vide.

```
let derive_clause_vide (lc:clause list) :bool =
  let rec traitement (clauses_a_traiter:clause list)
                     (clauses_prouvables:clause list) =
    match clauses_a_traiter with
    |[] -> clauses_prouvables
    |t::q -> begin
        let clauses_candidates = List.flatten
           (List.map (fun c -> (nouvelles_clauses c t)) clauses_prouvables) in
        let nouvelles_clauses = List.filter
            (fun c -> not (List.mem c clauses_a_traiter)
                    && not (List.mem c clauses_prouvables)) clauses_candidates in
        traitement (nouvelles_clauses@q) (t::clauses_prouvables)
      end
      in exists_clause_vide (traitement lc [])
```

   La fonction List.flatten permet "d'aplatir" la liste en entrée de sorte à transformer une liste de listes en une liste. Par exemple, List.flatten [[1;2;3];[2;5];[];[8]] = [1;2;3;2;5;8].

8. On remarque que la clause vide ne peut être obtenue que par coupure entre deux clauses de taille 1. Une stratégie pour essayer d'obtenir rapidement la clause vide serait donc d'appliquer la règle de coupure aux petites clauses prioritairement. Pour ce faire, on pourrait transformer les listes clauses_a_traiter et clauses_prouvables en files de priorité, la priorité d'une clause étant sa taille. Ainsi, les coupures suceptibles de produire rapidement la clause vide seraient faites en premier.

9. a) Laissé en exercice.

   b) Cela tient du fait que si plusieurs coupures sont possibles entre C_1 et C_2, les clauses résultantes de ces coupures seront toutes tautologiques donc pourront être loisiblement ignorées d'après 9a). En effet, dans le cas où plusieurs coupures sont possibles, on a sans perte de généralité p et q qui interviennent positivement dans C_1 et négativement dans C_2.

   Dans ces conditions, couper sur p produit une clause faisant intervenir q et $\neg q$ c'est-à-dire une clause tautologique et un raisonnement similaire tient si on coupe plutôt sur q.

   Ainsi, si C_1 et C_2 peuvent produire une clause par coupure, soit elles n'en produisent qu'une, soit toutes celles qu'elles produisent sont tautologiques et il suffit donc de couper sur une seule

variable pour obtenir toutes les clauses utiles à la construction d'une réfutation.

c) Voir le code source TP2_sans_totos.ml

10. On introduit les variables propositionnelles suivantes :

1 = être écossais.

2 = avoir des chaussettes rouges.

3 = porter un kilt.

4 = être marié.

5 = sortir le dimanche.

En utilisant le même formalisme que celui introduit pour les clauses, la formule $\neg i$ sera notée $-i$. Si on note F_i une formule modélisant la règle (i), on obtient :

$F_1 = -1 \Rightarrow 2 \equiv 1 \vee 2$ $\quad$ $F_4 = 1 \Leftrightarrow 5 \equiv (-1 \vee 5) \wedge (-5 \vee 1)$

$F_2 = 2 \Rightarrow 3 \equiv -2 \vee 3$ $\quad$ $F_5 = 3 \Rightarrow (1 \wedge 4) \equiv (-3 \vee 1) \wedge (-3 \vee 4)$

$F_3 = 4 \Rightarrow -5 \equiv -4 \vee -5$ $\quad$ $F_6 = 1 \Rightarrow 3 \equiv -1 \vee 3$

11. La question précédente montre qu'on peut entrer dans le club écossais si et seulement si l'ensemble de clauses $S = \{1 \vee 2, -2 \vee 3, -4 \vee -5, -1 \vee 5, -5 \vee 1, -3 \vee 1, -3 \vee 4, -1 \vee 3\}$ est satisfiable.

On utilise la fonction derive_clause_vide sur cet ensemble de clauses et on constate qu'on obtient true comme résultat. Puisque derive_clause_vide est correcte (d'après la partie 3), on en déduit que S admet une réfutation par coupure et la correction de la réfutation par coupure (voir partie 3) assure alors que S n'est pas satisfiable donc que personne ne peut entrer dans le club.

Exercice (***) : Comment pourrait-on modifier cette fonction afin qu'elle fournisse une réfutation par coupure de son entrée si cette réfutation existe ? Comment pourrait-on la modifier pour qu'un humain puisse interagir afin de proposer des coupures à faire prioritairement ; la fonction derive_clause_vide se contentant alors de mettre à jour les clauses qui ont déjà été obtenues ?

12. a) D'après l'hypothèse pesant sur C_1, C_2 et C, on peut loisiblement supposer qu'il existe une variable v intervenant positivement dans C_1, négativement dans C_2 et telle que C se déduit par coupure sur v de C_1 et C_2. Soit φ une valuation satisfaisant C_1 et C_2. Supposons par l'absurde qu'elle ne satisfait pas C. Alors d'après la sémantique standard de $\vee$ on a :

(1) Toute variable positive w intervenant dans C, vérifie $\varphi(w) = 0$. Vu la définition de la règle de coupure, cela implique que toutes les variables positives de C_1 sauf éventuellement v sont rendues fausses par φ, de même que toutes les variables positives de C_2.

(2) Toute variable négative w intervenant dans C, vérifie $\varphi(w) = 1$. Vu la définition de la règle de coupure, cela implique que toutes les variables négatives de C_2 sauf éventuellement v sont rendues vraies par φ, de même que toutes les variables négatives de C_1.

Si $\varphi(v) = 1$, alors toutes les variables positives de C_2 sont rendues fausses par φ par (1) et toutes les variables négatives de C_2 sont rendues vraies par φ par (2), donc $\varphi(C_2) = 0$. Si $\varphi(v) = 0$, on obtient similairement $\varphi(C_1) = 0$. Dans les deux cas on obtient une contradiction avec le fait que φ satisfait $\{C_1, C_2\}$ ce qui permet de conclure que $\{C_1, C_2\} \models C$.

b) On procède par induction. Si $S \vdash C$ on est par définition dans l'un des cas suivants :

- $F \in C$ auquel cas on a évidemment $S \models F$.

- Il existe C_1, C_2 telles que $S \vdash C_1$, $S \vdash C_2$ et C se déduit par coupure de C_1 et C_2. Si φ satisfait S, l'hypothèse inductive assure qu'elle satisfait C_1 et C_2 et la question précédente qu'elle satisfait alors C donc $S \models C$ aussi.

c) Si $S \vdash \bot$, la question précédente montre que $S \models \bot$, autrement dit que toute valuation satisfaisant S satisfait aussi $\bot$: cette dernière formule étant antilogique, aucune valuation ne satisfait S. On vient de montrer que tout ensemble réfutable par coupure est non satisfiable.

13. a) Si S ne contient pas la clause vide, il contient au moins deux clauses car une seule clause non vide est toujours satisfiable. Supposons par l'absurde qu'il n'existe aucune variable apparaissant positivement dans l'une des clauses de S et négativement dans une autre. Dans ces conditions, la valuation attribuant la valeur vrai à toutes les variables positives impliquées dans S et faux à celles négatives satisfait S ce qui est une contradiction.

b) On procède par double implication.

Si S est satisfiable, Σ_w aussi par correction de la règle de coupure prouvée en 12.a).

Réciproquement, soit φ une valuation satisfaisant Σ_w. Prolongeons φ de sorte à ce qu'elle satisfasse S. Deux cas se présentent :

- Il existe une clause $C = C_1 \vee w \in S_w$ telle que $\varphi(C_1) = 0$. Alors, on prolonge φ en $\bar{\varphi}$ de telle sorte à ce que $\bar{\varphi}(w) = 1$. Toutes les clauses de S_w faisant intervenir w positivement sont évidemment satisfaites par $\bar{\varphi}$. De plus, si $C' = C_2 \vee \neg w \in S_w$ fait intervenir w négativement, on sait que $C_1 \vee C_2 \in \mathrm{Res}(S_w)$ — puisque cette clause est obtenable par coupure sur w à partir de C et C' — donc est satisfaite par φ. Mais par hypothèse, $\varphi(C_1) = 0$ donc nécessairement $\varphi(C_2) = 1$ et donc $\bar{\varphi}(C') = 1$ aussi. Finalement, $\bar{\varphi}$ satisfait toutes les clauses de S_w et comme elle satisfait par hypothèse les clauses de $S \setminus S_w$, elle satisfait S.
- Pour toute clause $C = C_1 \vee w \in S_w$ on a $\varphi(C_1) = 1$. Alors on prolonge φ en $\bar{\varphi}$ telle que $\bar{\varphi}(w) = 0$ et un raisonnement similaire montre que cette valuation satisfait S.

Dans tous les cas, S est effectivement satisfiable et l'équivalence est avérée.

c) Montrons par récurrence forte sur $n \in \mathbb{N}^\star$ que : "Tout ensemble fini et non satisfiable de clauses faisant intervenir n variables est réfutable par coupure".

Si un tel ensemble ne fait intervenir qu'une variable, soit il contient la clause vide et est donc réfutable par coupure, soit pas et alors la question 13.a) montre que S contient deux clauses de la forme v et $\neg v$ à partir desquelles on peut déduire la clause vide par coupure.

Si S est un ensemble fini non satisfiable de clauses faisant intervenir $n+1$ variables, on choisit une variable w intervenant dans S et la question 13.b) montre que Σ_w est un ensemble de clauses non satisfiable. De plus, par construction, cet ensemble contient au plus n variables puisque w n'y intervient plus. On peut donc lui appliquer l'hypothèse de récurrence et conclure à sa réfutabilité par coupure puis à celle de S.

d) La question 13.c) et le théorème de compacité concluent sans autre forme de procès.

14. Il s'agit de montrer que la liste C_t finit par être vide. C'est effectivement le cas car :

- La liste C_t ne contient que des clauses simplifiées prouvables par coupure à partir de l'ensemble de clauses en entrée S (par correction de coupure).
- Une clause simplifiée et prouvable par coupure à partir de S n'est ajoutée qu'une seule fois à la liste C_t étant donné le filtrage effectué sur les potentielles nouvelles clauses.

- Le nombre de clauses simplifiées prouvables par coupure à partir d'un ensemble de clauses faisant intervenir n variables est majoré par 2^{2n} (chacun des $2n$ littéraux intervient ou pas dans chacune des clauses).

Autrement dit, on ajoute au plus une fois chacun des éléments d'un ensemble fini à C_t : comme on retire un élément de C_t à chaque itération, cette liste finit bien par être vide.

On reformule l'idée de l'algorithme sous-jacent à derive_clause_vide via le pseudo-code suivant :

```
derive_clause_vide(S) =
  C_t ← S    //C_t représente les clauses à traiter
  C_p ← ∅    //C_p contiendra à la fin toutes les clauses prouvables
  Tant que C_t ≠ ∅
    c ← une des clauses de C_t
    D_c ← ∅
    Pour toute clause c' ∈ C_p
      D_c ← D_c ∪ {clauses déductibles en une étape à partir de c et c'}
    C_t ← C_t ∪ {clauses de D_c n'appartenant ni à C_t ni à C_p}
    C_p ← C_p ∪ {c}
  Si la clause vide est dans C_p renvoyer vrai, sinon faux
```

15. L'invariant proposé par l'énoncé est vrai avant de rentrer dans la boucle tant que de l'algorithme ci-dessous d'après les merveilleuses propriétés de l'ensemble vide.

    Supposons que toutes les clauses déductibles à partir de C_p sont soit dans C_t soit dans C_p avant une itération et notons C'_t et C'_p le contenu de ces variables avant la suivante. D'après l'algorithme, $C'_p = C_p \cup \{c\}$ où c est une clause de C_p et $C'_t = C_t \cup \{\text{clauses déductibles à partir d'une clause de } C_p \text{ et de } c \text{ n'appartenant ni à } C_t \text{ ni à } C_p\}$.

    Les clauses déductibles à partir de C_p sont déjà dans C'_p ou C'_t par hypothèse et les clauses déductibles à partir de c et l'une des clauses de C_p étaient soit déjà dans $C_p \cup C_t \subset C'_p \cup C'_t$, soit sont ajoutées à C_t pour former C'_t. Donc toute clause déductible à partir C'_p est dans C'_p ou C'_t, comme convenu.

    En particulier en fin d'algorithme, comme C_t est vide, toutes les clauses déductibles à partir de C_p sont dans C_p : l'ensemble des clauses prouvables par coupure est saturé.

16. Par construction et correction de coupure, C_p est inclus dans l'ensemble des clauses prouvables à partir de S. Montrons l'inclusion inverse par induction. Si C est prouvable à partir de S on est dans un des deux cas suivants :

    - $C \in S$. Comme $C_t = S$ initialement et est vide à la fin, toutes les clauses de S sont dans C_p à la fin de l'algorithme donc en particulier C.
    - C se déduit par coupure (en une étape) à partir de deux clauses C_1 et C_2 prouvables à partir de S. Par hypothèse, ces deux clauses sont dans C_p en fin d'algorithme et la question 15 montre que C y est donc aussi.

    On en déduit que C_p est exactement l'ensemble des clauses prouvables à partir de S donc la clause vide est prouvable à partir de S si et seulement si elle est présente dans C_p en fin d'algorithme ce qui montre la correction de derive_clause_vide.